9

Language Integrated Query

WHAT'S IN THIS CHAPTER?

- Working with traditional queries across objects using List
- Using extension methods
- Getting to know LINQ query operators
- Working with Parallel LINQ
- Working with expression trees

CODE DOWNLOADS FOR THIS CHAPTER

The source code for this chapter is available on the book page at `www.wiley.com`. Click the Downloads link. The code can also be found at `https://github.com/ProfessionalCSharp/ProfessionalCSharp2021` in the directory `1_CS/LINQ`.

The code for this chapter is divided into the following major examples:

- LINQIntro
- EnumerableSample
- ParallelLINQ
- ExpressionTrees

All the sample projects have nullable reference types configured.

LINQ OVERVIEW

Language Integrated Query (LINQ) integrates query syntax inside the C# programming language, making it possible to access different data sources with the same syntax. LINQ accomplishes this by offering an abstraction layer.

This chapter describes the core principles of LINQ and the language extensions for C# that make the C# LINQ query possible.

> **NOTE** *For details about using LINQ across a database, read Chapter 21, "Entity Framework Core."*

This chapter starts with a simple LINQ query before diving into the full potential of LINQ. The C# language offers an integrated query language that is converted to method calls. This section shows you what the conversion looks like so you can use all the possibilities of LINQ.

Lists and Entities

The LINQ queries in this chapter are performed on a collection containing Formula 1 champions from 1950 to 2020. This data needs to be prepared with classes and lists within a .NET 5.0 library.

For the entities, the record type `Racer` is defined (as shown in the following code snippet). `Racer` defines several properties and an overloaded `ToString` method to display a racer in a string format. This class implements the interface `IFormattable` to support different variants of format strings, and the interface `IComparable<Racer>`, which can be used to sort a list of racers based on the `LastName`. For more advanced queries, the class `Racer` contains not only single-value properties such as `FirstName`, `LastName`, `Wins Country`, and `Starts`, but also properties that contain a collection, such as `Cars` and `Years`. The `Years` property lists all the years of the championship title. Some racers have won more than one title. The `Cars` property is used to list all the cars used by the driver during the title years (code file `DataLib/Racer.cs`):

```
public record Racer(string FirstName, string LastName, string Country,
  int Starts, int Wins, IEnumerable<int> Years, IEnumerable<string> Cars) :
  IComparable<Racer>, IFormattable
{
  public Racer(string FirstName, string LastName, string Country,
    int Starts, int Wins)
    : this(FirstName, LastName, Country, Starts, Wins, new int[] { },
      new string[] { })
  { }

  public override string ToString() => $"{FirstName} {LastName}";

  public int CompareTo(Racer? other) => LastName.CompareTo(other?.LastName);

  public string ToString(string format) => ToString(format, null);

  public string ToString(string? format, IFormatProvider? formatProvider) =>
    format switch
    {
      null => ToString(),
      "N" => ToString(),
      "F" => FirstName,
```

```
        "L" => LastName,
        "C" => Country,
        "S" => Starts.ToString(),
        "W" => Wins.ToString(),
        "A" => $"{FirstName} {LastName}, country: {Country}, starts: {Starts},
          wins: {Wins}",
        _ => throw new FormatException($"Format {format} not supported")
      };
    }
}
```

> **NOTE** *With the Formula 1 racing series, in every calendar year, a driver championship and a constructor championship take place. With the driver championship, the best driver is world champion. With the constructor championship, the best team wins the award. See* `https://www.formula1.com` *for details, current standings, and an archive going back to 1950.*

A second entity class is `Team`. This class just contains the team name and an array of years for constructor championships (code file `DataLib/Team.cs`):

```
public record Team
{
  public Team(string name, params int[] years)
  {
    Name = name;
    Years = years != null ? new List<int>(years) : new List<int>();
  }
  public string Name { get; }
  public IEnumerable<int> Years { get; }
}
```

The class `Formula1` returns a list of racers in the method `GetChampions`. The list is filled with all Formula 1 champions from the years 1950 to 2020 with the method `InitializeRacers` (code file `DataLib/Formula1.cs`):

```
public static class Formula1
{
  private static List<Racer> s_racers;
  public static IList<Racer> GetChampions() => s_racers ??= InitalizeRacers();

  private static List<Racer> InitializeRacers => new()
  {
    new ("Nino", "Farina", "Italy", 33, 5, new int[] { 1950 },
      new string[] { "Alfa Romeo" }),
    new ("Alberto", "Ascari", "Italy", 32, 10, new int[] { 1952, 1953 },
      new string[] { "Ferrari" }),
    new ("Juan Manuel", "Fangio", "Argentina", 51, 24,
      new int[] { 1951, 1954, 1955, 1956, 1957 },
      new string[] { "Alfa Romeo", "Maserati", "Mercedes", "Ferrari" }),
    new ("Mike", "Hawthorn", "UK", 45, 3, new int[] { 1958 },
      new string[] { "Ferrari" }),
    new ("Phil", "Hill", "USA", 48, 3, new int[] { 1961 },
      new string[] { "Ferrari" }),
    new ("John", "Surtees", "UK", 111, 6, new int[] { 1964 },
```

```
        new string[] { "Ferrari" }),
      new ("Jim", "Clark", "UK", 72, 25, new int[] { 1963, 1965 },
        new string[] { "Lotus" }),
      //...
    };
    //...
  }
```

Where queries are done across multiple lists, the `GetConstructorChampions` method in the following code snippet returns the list of all constructor championships (these championships have been around since 1958):

```
private static List<Team> s_teams;
public static IList<Team> GetConstructorChampions() => s_teams ??= new()
{
  new ("Vanwall", 1958),
  new ("Cooper", 1959, 1960),
  new ("Ferrari", 1961, 1964, 1975, 1976, 1977, 1979, 1982, 1983, 1999,
      2000, 2001, 2002, 2003, 2004, 2007, 2008),
  new ("BRM", 1962),
  new ("Lotus", 1963, 1965, 1968, 1970, 1972, 1973, 1978),
  new ("Brabham", 1966, 1967),
  new ("Matra", 1969),
  new ("Tyrrell", 1971),
  new ("McLaren", 1974, 1984, 1985, 1988, 1989, 1990, 1991, 1998),
  new ("Williams", 1980, 1981, 1986, 1987, 1992, 1993, 1994, 1996, 1997),
  new ("Benetton", 1995),
  new ("Renault", 2005, 2006),
  new ("Brawn GP", 2009),
  new ("Red Bull Racing", 2010, 2011, 2012, 2013),
  new ("Mercedes", 2014, 2015, 2016, 2017, 2018, 2019, 2020)
};
```

LINQ Query

Using these prepared lists and objects from the previously created library, you can do a LINQ query—for example, a query to get all world champions from Brazil sorted by the highest number of wins. To accomplish this, you could use methods of the `List<T>` class—for example, the `FindAll` and `Sort` methods. However, with LINQ there's a simpler syntax (code file `LINQIntro/Program.cs`):

```
static void LinqQuery()
{
  var query = from r in Formula1.GetChampions()
              where r.Country == "Brazil"
              orderby r.Wins descending
              select r;

  foreach (Racer r in query)
  {
    Console.WriteLine($"{r:A}");
  }
}
```

The result of this query shows world champions from Brazil ordered by number of wins:

```
Ayrton Senna, country: Brazil, starts: 161, wins: 41
Nelson Piquet, country: Brazil, starts: 204, wins: 23
Emerson Fittipaldi, country: Brazil, starts: 143, wins: 14
```

The expression

```
from r in Formula1.GetChampions()
where r.Country == "Brazil"
orderby r.Wins descending
select r;
```

is a LINQ query. The clauses `from`, `where`, `orderby`, `descending`, and `select` are predefined keywords in this query.

The query expression must begin with a `from` clause and end with a `select` or `group` clause. In between, you can optionally use `where`, `orderby`, `join`, `let`, and additional `from` clauses.

> **NOTE** *The variable* `query` *just has the LINQ query assigned to it. The query is not performed by this assignment but rather as soon as the query is accessed using the* `foreach` *loop. This is discussed in more detail later in the section "Deferred Query Execution."*

Extension Methods

The compiler converts the LINQ query to method calls. At runtime, extension methods will be invoked. LINQ offers various extension methods for the `IEnumerable<T>` interface, so you can use the LINQ query across any collection that implements this interface. An extension method is defined as a static method whose first parameter defines the type it extends, and it is declared in a static class.

> **NOTE** *Extension methods are covered in Chapter 3, "Classes, Records, Structs, and Tuples."*

One of the classes that define LINQ extension methods is `Enumerable` in the namespace `System.Linq`. You just have to import the namespace to open the scope of the extension methods of this class. A sample implementation of the `Where` extension method is shown in the following code. The first parameter of the `Where` method that includes the `this` keyword is of type `IEnumerable<T>`. This enables the `Where` method to be used with every type that implements `IEnumerable<T>`. A few examples of types that implement this interface are arrays and `List<T>`. The second parameter is a `Func<T, bool>` delegate that references a method that returns a Boolean value and requires a parameter of type `T`. This predicate is invoked within the implementation to examine whether the item from the `IEnumerable<T>` source should be added into the destination collection. If the method is referenced by the delegate, the `yield return` statement returns the item from the source to the destination:

```
public static IEnumerable<TSource> Where<TSource>(
  this IEnumerable<TSource> source,
  Func<TSource, bool> predicate)
{
  foreach (TSource item in source)
  {
    if (predicate(item))
      yield return item;
  }
}
```

Because `Where` is implemented as a generic method, it works with any type that is contained in a collection. Any collection implementing `IEnumerable<T>` is supported.

Now it's possible to use the extension methods `Where`, `OrderByDescending`, and `Select` from the class `Enumerable`. Because each of these methods returns `IEnumerable<TSource>`, it is possible to invoke one method after the other by using the previous result. With the arguments of the extension methods, anonymous methods that define the implementation for the delegate parameters are used (code file `LINQIntro/Program.cs`):

```
static void ExtensionMethods()
{
  List<Racer> champions = new(Formula1.GetChampions());
  var brazilChampions =
    champions.Where(r => r.Country == "Brazil")
      .OrderByDescending(r => r.Wins)
      .Select(r => r);

  foreach (Racer r in brazilChampions)
  {
    Console.WriteLine($"{r:A}");
  }
}
```

Deferred Query Execution

During runtime, the query expression does not run immediately as it is defined. The query runs only when the items are iterated. The reason is that the extension method shown earlier makes use of the `yield return` statement to return the elements where the predicate is true. Because the `yield return` statement is used, the compiler creates an enumerator and returns the items as soon as they are accessed from the enumeration.

This has a very interesting and important effect. In the following example, a collection of `string` elements is created and filled with first names. Next, a query is defined to get all the names from the collection whose first letter is `J`. The collection should also be sorted. The iteration does not happen when the query is defined. Instead, the iteration happens with the `foreach` statement, where all items are iterated. Only one element of the collection fulfills the requirements of the `where` expression to start with the letter J: `Juan`. After the iteration is done and `Juan` is written to the console, four new names are added to the collection. Then the iteration is done again (code file `LINQIntro/Program.cs`):

```
void DeferredQuery()
{
  List<string> names = new() { "Nino", "Alberto", "Juan", "Mike", "Phil" };
  var namesWithJ = from n in names
                   where n.StartsWith("J")
                   orderby n
                   select n;

  Console.WriteLine("First iteration");
  foreach (string name in namesWithJ)
  {
    Console.WriteLine(name);
  }
  Console.WriteLine();

  names.Add("John");
  names.Add("Jim");
  names.Add("Jack");
  names.Add("Denny");
  Console.WriteLine("Second iteration");
```

```
  foreach (string name in namesWithJ)
  {
    Console.WriteLine(name);
  }
}
```

Because the iteration does not happen when the query is defined, but does happen with every `foreach`, the output from the application changes:

```
First iteration
Juan
Second iteration
Jack
Jim
John
Juan
```

Of course, you also must be aware that the extension methods are invoked every time the query is used within an iteration. Most of the time, this is very practical because you can detect changes in the source data. However, sometimes this is impractical. You can change this behavior by invoking the extension methods `ToArray`, `ToList`, and the like. In the following example, you can see that `ToList` iterates through the collection immediately and returns a collection implementing `IList<string>`. The returned list is then iterated through twice; in between iterations, the data source gets new names:

```
List<string> names = new() { "Nino", "Alberto", "Juan", "Mike", "Phil" };
var namesWithJ = (from n in names
                  where n.StartsWith("J")
                  orderby n
                  select n).ToList();

Console.WriteLine("First iteration");
foreach (string name in namesWithJ)
{
  Console.WriteLine(name);
}
Console.WriteLine();

names.Add("John");
names.Add("Jim");
names.Add("Jack");
names.Add("Denny");

Console.WriteLine("Second iteration");
foreach (string name in namesWithJ)
{
  Console.WriteLine(name);
}
```

The result indicates that in between the iterations, the output stays the same although the collection values have changed:

```
First iteration
Juan
Second iteration
Juan
```

STANDARD QUERY OPERATORS

`Where`, `OrderByDescending`, and `Select` are only a few of the query operators defined by LINQ. The LINQ query defines a declarative syntax for the most common operators. There are many more query operators available with the `Enumerable` class.

The following table lists the standard query operators defined by the `Enumerable` class.

STANDARD QUERY OPERATORS	DESCRIPTION
`Where` `OfType<TResult>`	*Filtering operators* define a restriction to the elements returned. With the `Where` query operator, you can use a predicate; for example, a lambda expression that returns a bool. `OfType<TResult>` filters the elements based on the type and returns only the elements of the type `TResult`.
`Select` `SelectMany`	*Projection operators* are used to transform an object into a new object of a different type. `Select` and `SelectMany` define a projection to select values of the result based on a selector function.

`OrderBy`
`ThenBy`
`OrderByDescending`
`ThenByDescending`
`Reverse`
	Sorting operators change the order of elements returned. `OrderBy` sorts values in ascending order. `OrderByDescending` sorts values in descending order. `ThenBy` and `ThenByDescending` operators are used for a secondary sort if the first sort gives similar results. `Reverse` reverses the elements in the collection.

`Join` `GroupJoin`	*Join operators* are used to combine collections that might not be directly related to each other. With the `Join` operator, you can do a join of two collections based on key selector functions. This is similar to the JOIN you know from SQL. The `GroupJoin` operator joins two collections and groups the results.
`GroupBy` `ToLookup`	*Grouping operators* put the data into groups. The `GroupBy` operator groups elements with a common key. `ToLookup` groups the elements by creating a one-to-many dictionary.

`Any`
`All`
`Contains`
	Quantifier operators return a Boolean value if elements of the sequence satisfy a specific condition. `Any`, `All`, and `Contains` are quantifier operators. `Any` determines whether any element in the collection satisfies a predicate function. `All` determines whether all elements in the collection satisfy a predicate. `Contains` checks whether a specific element is in the collection.

`Take`
`Skip`
`TakeWhile`
`SkipWhile`
	Partitioning operators return a subset of the collection. `Take`, `Skip`, `TakeWhile`, and `SkipWhile` are partitioning operators. With these, you get a partial result. With `Take`, you have to specify the number of elements to take from the collection. `Skip` ignores the specified number of elements and takes the rest. `TakeWhile` takes the elements as long as a condition is true. `SkipWhile` skips the elements as long as the condition is true.

STANDARD QUERY OPERATORS	DESCRIPTION

`Distinct`
`Union`
`Intersect`
`Except`
`Zip`
	Set operators return a collection set. `Distinct` removes duplicates from a collection. With the exception of `Distinct`, the other set operators require two collections. `Union` returns unique elements that appear in either of the two collections. `Intersect` returns elements that appear in both collections. `Except` returns elements that appear in just one collection. `Zip` combines two collections into one.

`First`
`FirstOrDefault`
`Last`
`LastOrDefault`
`ElementAt`
`ElementAtOrDefault`
`Single`
`SingleOrDefault`
	Element operators return just one element. `First` returns the first element that satisfies a condition. `FirstOrDefault` is similar to `First`, but it returns a default value of the type if the element is not found. `Last` returns the last element that satisfies a condition. With `ElementAt`, you specify the position of the element to return. `Single` returns only the one element that satisfies a condition. If more than one element satisfies the condition, an exception is thrown. All the `XXOrDefault` methods are similar to the methods that start with the same prefix, but they return the default value of the type if the element is not found.

`Count`
`Sum`
`Min`
`Max`
`Average`
`Aggregate`
	Aggregate operators compute a single value from a collection. With aggregate operators, you can get the sum of all values, the number of all elements, the element with the lowest or highest value, an average number, and so on.

`ToArray`
`AsEnumerable`
`ToList`
`ToDictionary`
`Cast<TResult>`
	Conversion operators convert the collection to an array: `IEnumerable`, `IList`, `IDictionary`, and so on. The `Cast` method casts every item of the collection to the generic argument type.

`Empty`
`Range`
`Repeat`
	Generation operators return a new sequence. The `Empty` operator returns an empty `IEnumerable`, `Range` returns `IEnumerable` containing a sequence of numbers, and `Repeat` returns `IEnumerable` with one repeated value.

The following sections provide examples demonstrating how to use these operators.

Filter

This section looks at some examples for a query. This sample application available with the code download offers passing command-line arguments for every different feature shown. With the Debug section in the Properties of Visual Studio, you can configure the command-line arguments as needed to run the different sections of the

application. Using the command line with the installed SDK, you can invoke the commands using .NET CLI in this way:

```
> dotnet run -- filter simplefilter
```

which passes the arguments `filter simplefilter` to the application.

With the `where` clause, you can combine multiple expressions—for example, get only the racers from Brazil and Austria who won more than 15 races. The result type of the expression passed to the `where` clause just needs to be of type `bool` (code file `EnumerableSample/FilterSamples.cs`):

```
public static void SimpleFilter()
{
  var racers = from r in Formula1.GetChampions()
               where r.Wins > 15 &&
               (r.Country == "Brazil" || r.Country == "Austria")
               select r;

  foreach (var r in racers)
  {
    Console.WriteLine($"{r:A}");
  }
}
```

Starting the program with this LINQ query (`filter simplefilter`) returns Niki Lauda, Nelson Piquet, and Ayrton Senna, as shown here:

```
Niki Lauda, country: Austria, Starts: 173, Wins: 25
Nelson Piquet, country: Brazil, Starts: 204, Wins: 23
Ayrton Senna, country: Brazil, Starts: 161, Wins: 41
```

Not all queries can be done with the LINQ query syntax, and not all extension methods are mapped to LINQ query clauses. Advanced queries require using extension methods. To better understand complex queries with extension methods, it's good to see how simple queries are mapped. The following code uses the `Where` extension method instead of a LINQ query. The `Select` extension method would return the same object returned by the `Where` method, so it isn't needed here (code file `EnumerableSample/FilterSamples.cs`):

```
public static void FilterWithMethods()
{
  var racers = Formula1.GetChampions()
    .Where(r => r.Wins > 15 &&
      (r.Country == "Brazil" || r.Country == "Austria"));
  //...
}
```

Filter with Index

One scenario in which you can't use the LINQ query is an overload of the `Where` method. With an overload of the `Where` method, you can pass a second parameter that is the index. The index is a counter for every result returned from the filter. You can use the index within the expression to do some calculation based on the index. In the following example, the index is used within the code that is called by the `Where` extension method to return only racers whose last name starts with A if the index is even (code file `EnumerableSample/FilterSamples.cs`):

```
public static void FilteringWithIndex()
{
  var racers = Formula1.GetChampions()
    .Where((r, index) => r.LastName.StartsWith("A") && index % 2 != 0);
```

```
    foreach (var r in racers)
    {
      Console.WriteLine($"{r:A}");
    }
  }
```

The racers with last names beginning with the letter A are Alberto Ascari, Mario Andretti, and Fernando Alonso. Because Mario Andretti is positioned within an index that is odd, he is not in the result:

```
Alberto Ascari, Italy; starts: 32, wins: 13
Fernando Alonso, Spain; starts: 314, wins: 32
```

Type Filtering

For filtering based on a type, you can use the `OfType` extension method. Here the array data contains both `string` and `int` objects. When you use the extension method `OfType`, passing the string class to the generic parameter returns only the strings from the collection (code file `EnumerableSample/FilterSamples.cs`):

```
public static void TypeFilter()
{
  object[] data = { "one", 2, 3, "four", "five", 6 };
  var query = data.OfType<string>();

  foreach (var s in query)
  {
    Console.WriteLine(s);
  }
}
```

When you run this code, the strings one, four, and five are displayed:

```
one
four
five
```

Compound from

If you need to do a filter based on a member of the object that itself is a sequence, you can use a compound `from`. The `Racer` class defines a property `Cars`, where `Cars` is a string array. For a filter of all racers who were champions with a Ferrari, you can use the LINQ query shown next. The first `from` clause accesses the `Racer` objects returned from `Formula1.GetChampions`. The second `from` clause accesses the `Cars` property of the `Racer` class to return all cars of type `string`. Next the cars are used with the `where` clause to filter only the racers who were champions with a Ferrari (code file `EnumerableSample/CompoundFromSamples.cs`):

```
public static void CompoundFrom()
{
  var ferrariDrivers = from r in Formula1.GetChampions()
                       from c in r.Cars
                       where c == "Ferrari"
                       orderby r.LastName
                       select r.FirstName + " " + r.LastName;
  //...
}
```

If you are curious about the result of this query, following are all Formula 1 champions driving a Ferrari:

```
Alberto Ascari
Juan Manuel Fangio
```

```
Mike Hawthorn
Phil Hill
Niki Lauda
Kimi Räikkönen
Jody Scheckter
Michael Schumacher
John Surtees
```

The C# compiler converts a compound `from` clause with a LINQ query to the `SelectMany` extension method. You can use `SelectMany` to iterate a sequence of a sequence. The overload of the `SelectMany` method that is used with the example is shown here:

```
public static IEnumerable<TResult> SelectMany<TSource, TCollection, TResult> (
  this IEnumerable<TSource> source,
  Func<TSource,
  IEnumerable<TCollection>> collectionSelector,
  Func<TSource, TCollection, TResult> resultSelector);
```

The first parameter is the implicit parameter that receives the sequence of `Racer` objects from the `GetChampions` method. The second parameter is the `collectionSelector` delegate where the inner sequence is defined. With the lambda expression `r => r.Cars`, the collection of cars should be returned. The third parameter is a delegate that is now invoked for every car and receives the `Racer` and `Car` objects. The lambda expression creates an anonymous type with a `Racer` and a `Car` property. As a result of this `SelectMany` method, the hierarchy of racers and cars is flattened, and a collection of new objects of an anonymous type for every car is returned.

This new collection is passed to the `Where` method so that only the racers driving a Ferrari are filtered. Finally, the `OrderBy` and `Select` methods are invoked (code file `EnumerableSample/CompoundFromSamples.cs`):

```
public static void CompoundFromWithMethods()
{
  var ferrariDrivers = Formula1.GetChampions()
    .SelectMany(r => r.Cars, (r, c) => new { Racer = r, Car = c })
    .Where(r => r.Car == "Ferrari")
    .OrderBy(r => r.Racer.LastName)
    .Select(r => $"{r.Racer.FirstName} {r.Racer.LastName}");
  //...
}
```

Resolving the generic `SelectMany` method to the types that are used here, the types are resolved as follows. In this case, the source is of type `Racer`, the filtered collection is a `string` array, and, of course, the name of the anonymous type that is returned is not known and is shown here as `TResult`:

```
public static IEnumerable<TResult> SelectMany<Racer, string, TResult> (
  this IEnumerable<Racer> source,
  Func<Racer, IEnumerable<string>> collectionSelector,
  Func<Racer, string, TResult> resultSelector);
```

Because the query was just converted from a LINQ query to extension methods, the result is the same as before.

Sorting

To sort a sequence, the `orderby` clause was used already. This section reviews the earlier example, now with the `orderby descending` clause. Here the racers are sorted based on the number of wins as specified by the key selector in descending order (code file `EnumerableSample/SortingSamples.cs`):

```
public static void SortDescending()
{
  var racers = from r in Formula1.GetChampions()
```

```
                where r.Country == "Brazil"
                orderby r.Wins descending
                select r;
  //...
}
```

The `orderby` clause is resolved to the `OrderBy` method, and the `orderby descending` clause is resolved to the `OrderByDescending` method:

```
public static void SortDescendingWithMethods()
{
  var racers = Formula1.GetChampions()
    .Where(r => r.Country == "Brazil")
    .OrderByDescending(r => r.Wins)
    .Select(r => r);
  //...
}
```

The `OrderBy` and `OrderByDescending` methods return `IOrderedEnumerable<TSource>`. This interface derives from the interface `IEnumerable<TSource>` but contains an additional method, `CreateOrderedEnumerable <TSource>`. This method is used for further ordering of the sequence. If two items are the same based on the key selector, ordering can continue with the `ThenBy` and `ThenByDescending` methods. These methods require an `IOrderedEnumerable<TSource>` to work on but return this interface as well. Therefore, you can add any number of `ThenBy` and `ThenByDescending` methods to sort the collection.

When using the LINQ query, you just add all the different keys (with commas) for sorting to the `orderby` clause. In the next example, the sort of all racers is done first based on country, next on last name, and finally on first name. The `Take` extension method that is added to the result of the LINQ query is used to return the first 10 results:

```
public static void SortMultiple()
{
  var racers = (from r in Formula1.GetChampions()
                orderby r.Country, r.LastName, r.FirstName
                select r).Take(10);
  //...
}
```

The sorted result is shown here:

```
Argentina: Fangio, Juan Manuel
Australia: Brabham, Jack
Australia: Jones, Alan
Austria: Lauda, Niki
Austria: Rindt, Jochen
Brazil: Fittipaldi, Emerson
Brazil: Piquet, Nelson
Brazil: Senna, Ayrton
Canada: Villeneuve, Jacques
Finland: Hakkinen, Mika
```

Doing the same with extension methods makes use of the `OrderBy` and `ThenBy` methods:

```
public static void SortMultipleWithMethods()
{
  var racers = Formula1.GetChampions()
    .OrderBy(r => r.Country)
    .ThenBy(r => r.LastName)
```

```
      .ThenBy(r => r.FirstName)
      .Take(10);
    //...
}
```

Grouping

To group query results based on a key value, you can use the `group` clause. Now the Formula 1 champions should be grouped by country, and the number of champions within a country should be listed. The clause `group r by r.Country into g` groups all the racers based on the `Country` property and defines a new identifier `g` that you can use later to access the group result information. In the following example, the result from the `group` clause is ordered based on the extension method `Count` that is applied on the group result; and if the count is the same, the ordering is done based on the key. This is the country because this was the key used for grouping. The `where` clause filters the results based on groups that have at least two items, and the `select` clause creates an anonymous type with the `Country` and `Count` properties (code file `EnumerableSample/GroupSamples.cs`):

```
public static void Grouping()
{
  var countries = from r in Formula1.GetChampions()
                  group r by r.Country into g
                  orderby g.Count() descending, g.Key
                  where g.Count() >= 2
                  select new
                  {
                    Country = g.Key,
                    Count = g.Count()
                  };

  foreach (var item in countries)
  {
    Console.WriteLine($"{item.Country, -10} {item.Count}");
  }
}
```

The result displays the collection of objects with the `Country` and `Count` properties:

```
UK 10
Brazil 3
Finland 3
Germany 3
Australia 2
Austria 2
Italy 2
USA 2
```

Doing the same with extension methods, the `groupby` clause is resolved to the `GroupBy` method. What's interesting with the declaration of the `GroupBy` method is that it returns an enumeration of objects implementing the `IGrouping` interface. The `IGrouping` interface defines the `Key` property, so you can access the key of the group after defining the call to this method:

```
public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(
  this IEnumerable<TSource> source, Func<TSource, TKey> keySelector);
```

The `group r by r.Country into g` clause is resolved to `GroupBy(r => r.Country)` and returns the group sequence. The group sequence is first ordered by the `OrderByDescending` method, then by the `ThenBy` method. Next, the `Where` and `Select` methods that you already know are invoked (code file `EnumerableSample/GroupSamples.cs`):

```
public static void GroupingWithMethods()
{
  var countries = Formula1.GetChampions()
    .GroupBy(r => r.Country)
    .OrderByDescending(g => g.Count())
    .ThenBy(g => g.Key)
    .Where(g => g.Count() >= 2)
    .Select(g => new
    {
      Country = g.Key,
      Count = g.Count()
    });
  //...
}
```

Variables Within the LINQ Query

With the LINQ query as it is written for grouping, the `Count` method is called multiple times. You can change this by using the `let` clause. `let` allows defining variables within the LINQ query (code file `EnumerableSample/GroupSamples.cs`):

```
public static void GroupingWithVariables()
{
  var countries = from r in Formula1.GetChampions()
                  group r by r.Country into g
                  let count = g.Count()
                  orderby count descending, g.Key
                  where count >= 2
                  select new
                  {
                    Country = g.Key,
                    Count = count
                  };
  //...
}
```

> **NOTE** *Why is it a bad idea to invoke the* `Count` *method multiple times on a LINQ query? Of course, it's always faster to cache the result of a method instead of calling it multiple items. With the implementation of the extension method* `Count` *that's based on the* `IEnumerable` *interface, you should also think about how this method can be implemented. With the members of the* `IEnumerable` *interface, it's possible to iterate through all the elements and count the number of items in the list. The longer the list is, the longer it takes.*

Using the method syntax, the `Count` method was invoked multiple times as well. To define extra data to pass to the next method (what is really done by the `let` clause), you can use the `Select` method to create anonymous types. Here, an anonymous type with `Group` and `Count` properties is created. A collection of items with these properties is passed to the `OrderByDescending` method where the sort is based on the `Count` property of this anonymous type:

```
public static void GroupingWithAnonymousTypes()
{
  var countries = Formula1.GetChampions()
```

```
      .GroupBy(r => r.Country)
      .Select(g => new { Group = g, Count = g.Count() })
      .OrderByDescending(g => g.Count)
      .ThenBy(g => g.Group.Key)
      .Where(g => g.Count >= 2)
      .Select(g => new
      {
        Country = g.Group.Key,
        Count = g.Count
      });
    //...
}
```

Be aware of the number of interim objects created based on the `let` clause or `Select` method. When you query through large lists, the number of objects created that need to be garbage collected later on can have a huge impact on performance.

Grouping with Nested Objects

If the grouped objects should contain nested sequences, you can do that by changing the anonymous type created by the `select` clause. With this example, the returned countries should contain not only the properties for the name of the country and the number of racers, but also a sequence of the names of the racers. This sequence is assigned by using an inner `from/in` clause assigned to the `Racers` property. The inner `from` clause is using the `g` group to get all racers from the group, order them by last name, and create a new string based on the first and last name (code file `EnumerableSample/GroupSamples.cs`):

```
public static void GroupingAndNestedObjects()
{
  var countries = from r in Formula1.GetChampions()
                  group r by r.Country into g
                  let count = g.Count()
                  orderby count descending, g.Key
                  where count >= 2
                  select new
                  {
                    Country = g.Key,
                    Count = count,
                    Racers = from r1 in g
                             orderby r1.LastName
                             select r1.FirstName + " " + r1.LastName
                  };

  foreach (var item in countries)
  {
    Console.WriteLine($"{item.Country, -10} {item.Count}");
    foreach (var name in item.Racers)
    {
      Console.Write($"{name}; ");
    }
    Console.WriteLine();
  }
}
```

Using extension methods, the inner Racer objects are created using the group variable `g` of type `IGrouping` where the `Key` property is the key for the grouping—the country in this case—and the items of a group can be accessed using the `Group` property:

```
public static void GroupingAndNestedObjectsWithMethods()
{
  var countries = Formula1.GetChampions()
    .GroupBy(r => r.Country)
    .Select(g => new
    {
      Group = g,
      Key = g.Key,
      Count = g.Count()
    })
    .OrderByDescending(g => g.Count)
    .ThenBy(g => g.Key)
    .Where(g => g.Count >= 2)
    .Select(g => new
    {
      Country = g.Key,
      Count = g.Count,
      Racers = g.Group.OrderBy(r => r.LastName)
        .Select(r => r.FirstName + " " + r.LastName)
    });
  //...
}
```

The output now lists all champions from the selected countries:

```
UK        10
Jenson Button; Jim Clark; Lewis Hamilton; Mike Hawthorn; Graham Hill;
Damon Hill; James Hunt; Nigel Mansell; Jackie Stewart; John Surtees;
Brazil     3
Emerson Fittipaldi; Nelson Piquet; Ayrton Senna;
Finland    3
Mika Hakkinen; Kimi Raikkonen; Keke Rosberg;
Germany    3
Nico Rosberg; Michael Schumacher; Sebastian Vettel;
Australia  2
Jack Brabham; Alan Jones;
Austria    2
Niki Lauda; Jochen Rindt;
Italy      2
Alberto Ascari; Nino Farina;
USA        2
Mario Andretti; Phil Hill;
```

Inner Join

You can use the `join` clause to combine two sources based on specific criteria. First, however, let's get two lists that should be joined. With Formula 1, there are drivers and constructor champions. The drivers are returned from the method `GetChampions`, and the constructors are returned from the method `GetConstructorChampions`. It would be interesting to get a list that lists the driver and the constructor champions for each year.

To do this, the first two queries for the racers and the teams are defined (code file `EnumerableSample/JoinSamples.cs`):

```
public static void InnerJoin()
{
  var racers = from r in Formula1.GetChampions()
               from y in r.Years
               select new
               {
                 Year = y,
                 Name = r.FirstName + " " + r.LastName
               };

  var teams = from t in Formula1.GetConstructorChampions()
              from y in t.Years
              select new
              {
                Year = y,
                Name = t.Name
              };
  //...
}
```

Using these two queries, a join is done based on the year of the driver champion and the year of the team champion with the `join` clause. The `select` clause defines a new anonymous type containing `Year`, `Racer`, and `Team` properties:

```
var racersAndTeams = (from r in racers
                      join t in teams on r.Year equals t.Year
                      select new
                      {
                        r.Year,
                        Champion = r.Name,
                        Constructor = t.Name
                      }).Take(10);
Console.WriteLine("Year World Champion\t\t   Constructor Title");

foreach (var item in racersAndTeams)
{
  Console.WriteLine($"{item.Year}: {item.Champion,-20} {item.Constructor}");
}
```

Of course, you can also combine this to just one LINQ query, but that's a matter of taste:

```
var racersAndTeams =
  (from r in
   from r1 in Formula1.GetChampions()
   from yr in r1.Years
   select new
   {
     Year = yr,
     Name = r1.FirstName + " " + r1.LastName
   }
   join t in
     from t1 in Formula1.GetConstructorChampions()
     from yt in t1.Years
     select new
```

```
      {
        Year = yt,
        Name = t1.Name
      }
    on r.Year equals t.Year
    orderby t.Year
    select new
    {
      Year = r.Year,
      Racer = r.Name,
      Team = t.Name
    }).Take(10);
```

Using extension methods, the racers and teams can be joined by invoking the `Join` method, passing the teams with the first argument to join them with the racers, specifying the key selectors for the outer and inner collections, and defining the result selector with the last argument (code file `EnumerableSample/JoinSamples.cs`):

```
static void InnerJoinWithMethods()
{
  var racers = Formula1.GetChampions()
    .SelectMany(r => r.Years, (r1, year) =>
    new
    {
      Year = year,
      Name = $"{r1.FirstName} {r1.LastName}"
    });

  var teams = Formula1.GetConstructorChampions()
    .SelectMany(t => t.Years, (t, year) =>
    new
    {
      Year = year,
      Name = t.Name
    });

  var racersAndTeams = racers.Join(
    teams,
    r => r.Year,
    t => t.Year,
    (r, t) =>
      new
      {
        Year = r.Year,
        Champion = r.Name,
        Constructor = t.Name
      }).OrderBy(item => item.Year).Take(10);
  //...
}
```

The output displays data from the anonymous type for the first 10 years in which both a driver and constructor championship took place:

```
Year  World Champion   Constructor Title
1958: Mike Hawthorn    Vanwall
1959: Jack Brabham     Cooper
1960: Jack Brabham     Cooper
```

```
1961: Phil Hill         Ferrari
1962: Graham Hill       BRM
1963: Jim Clark         Lotus
1964: John Surtees      Ferrari
1965: Jim Clark         Lotus
1966: Jack Brabham      Brabham
1967: Denny Hulme       Brabham
```

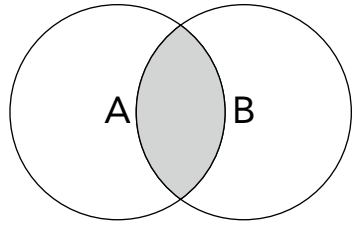

FIGURE 9-1

Figure 9-1 shows a graphical presentation of two collections combined with an inner join. Using an inner join, the results are matches with both collections.

Left Outer Join

The output from the previous join sample started with the year 1958—the first year when both the driver and constructor championships started. The driver championship had started earlier in the year 1950. With an inner join, results are returned only when matching records are found. To get a result with all the years included, you can use a left outer join. A *left outer join* returns all the elements in the left sequence even when no match is found in the right sequence.

The earlier LINQ query is changed to a left outer join. A left outer join is defined with the `join` clause together with the `DefaultIfEmpty` method. If the left side of the query (the racers) does not have a matching constructor champion, the default value for the right side is defined by the `DefaultIfEmpty` method (code file `EnumerableSample/JoinSamples.cs`):

```
public static void LeftOuterJoin()
{
  //...
  var racersAndTeams =
    (from r in racers
     join t in teams on r.Year equals t.Year into rt
     from t in rt.DefaultIfEmpty()
     orderby r.Year
     select new
     {
       Year = r.Year,
       Champion = r.Name,
       Constructor = t == null ? "no constructor championship" : t.Name
     }).Take(10);
  //...
}
```

When you do the same query with the extension methods, you use the `GroupJoin` method. The first three parameters are similar with `Join` and `GroupJoin`. The result of `GroupJoin` is different. Instead of a flat list that is returned from the `Join` method, `GroupJoin` returns a list where every matching item of the first list contains a list of matches from the second list. Using the following `SelectMany` method, the list is flattened again. In case no teams are available for a match, the `Constructors` property is assigned to the default value of the type, which is `null` with classes. Creating the anonymous type, the `Constructor` property gets the string "no constructor championship" assigned if the team is `null` (code file `EnumerableSample/JoinSamples.cs`):

```
public static void LeftOuterJoinWithMethods()
{
  //...
  var racersAndTeams =
    racers.GroupJoin(
      teams,
      r => r.Year,
```

```
        t => t.Year,
        (r, ts) => new
        {
          Year = r.Year,
          Champion = r.Name,
          Constructors = ts
        })
        .SelectMany(
          rt => rt.Constructors.DefaultIfEmpty(),
          (r, t) => new
          {
            Year = r.Year,
            Champion = r.Champion,
            Constructor = t?.Name ?? "no constructor championship"
          });
    //...
}
```

> **NOTE** *Other usages of the* `GroupJoin` *method are shown in the next section.*

When you run the application with this query, the output starts with the year 1950 as shown here:

```
Year Champion            Constructor Title
1950: Nino Farina        no constructor championship
1951: Juan Manuel Fangio no constructor championship
1952: Alberto Ascari     no constructor championship
1953: Alberto Ascari     no constructor championship
1954: Juan Manuel Fangio no constructor championship
1955: Juan Manuel Fangio no constructor championship
1956: Juan Manuel Fangio no constructor championship
1957: Juan Manuel Fangio no constructor championship
1958: Mike Hawthorn      Vanwall
1959: Jack Brabham       Cooper
```

Figure 9-2 shows a graphical presentation of two collections combined with a left outer join. When you use a left outer join, the results are not just matches with both collections A and B but also include the left collection A.

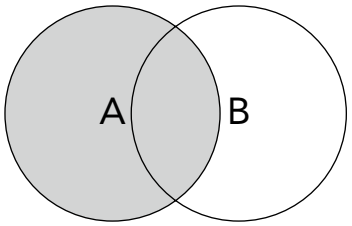

FIGURE 9-2

Group Join

A left outer join makes use of a group join together with the `into` clause. It uses partly the same syntax as the group join. The group join just doesn't need the `DefaultIfEmpty` method.

With a group join, two independent sequences can be joined, whereby one sequence contains a list of items for one element of the other sequence.

The following example uses two independent sequences. One is the list of champions that you already know from previous examples. The second sequence is a collection of `Championship` types. The `Championship` type is shown in the next code snippet. This class contains the year of the championship and the racers with the first, second, and third positions of the year with the properties `Year`, `First`, `Second`, and `Third` (code file `DataLib/Championship.cs`):

```
public record Championship(int Year, string First, string Second,
  string Third);
```

The collection of championships is returned from the method GetChampionships as shown in the following code snippet (code file DataLib/Formula1.cs):

```
private static List<Championship> s_championships;
public static IEnumerable<Championship> GetChampionships() =>
  s_championships ??= new()
  {
    new (1950, "Nino Farina", "Juan Manuel Fangio", "Luigi Fagioli"),
    new (1951, "Juan Manuel Fangio", "Alberto Ascari", "Froilan Gonzalez"),
    //...
  };
```

The list of champions should be combined with the list of racers that are found within the first three positions in every year of championships, and for every world champion the results for every year should be displayed.

Because in the list of championships every item contains three racers, this list needs to be flattened first. One way to do this is by using a compound from. As there's no collection available with a property of a single item, but instead the three properties First, Second, and Third need to be combined and flattened, a new List<T> is created that is filled with information from these properties. For a newly created object, custom classes and anonymous types can be used as you've already seen several times. This time, you'll create a tuple. Tuples contain members of different types and can be created using tuple literals with parentheses as shown in the following code snippet. Here, the code creates a flat list of tuples containing the year, position in the championship, first name, and last name information from racers (code file EnumerableSample/JoinSamples.cs):

```
static void GroupJoin()
{
  var racers = from cs in Formula1.GetChampionships()
               from r in new List<
                 (int Year, int Position, string FirstName, string LastName)>()
               {
                 (cs.Year, Position: 1, FirstName: cs.First.FirstName(),
                  LastName: cs.First.LastName()),
                 (cs.Year, Position: 2, FirstName: cs.Second.FirstName(),
                  LastName: cs.Second.LastName()),
                 (cs.Year, Position: 3, FirstName: cs.Third.FirstName(),
                  LastName: cs.Third.LastName())
               }
               select r;
  //...
}
```

The extension methods FirstName and LastName just use the last blank character to split up the string (code file EnumerableSample/StringExtensions.cs):

```
public static class StringExtensions
{
  public static string FirstName(this string name) =>
    name.Substring(0, name.LastIndexOf(' '));

  public static string LastName(this string name) =>
    name.Substring(name.LastIndexOf(' ') + 1);
}
```

With a join clause, the racers from both lists can be combined. Formula1.GetChampions returns a list of Racers, and the racers variable returns the list of tuples that contains the year, the result, and the names of racers. It's not enough to compare the items from these two collections by using the last name. Sometimes a racer and his father can be found in the list (for example, Damon Hill and Graham Hill), so it's necessary to compare the items by both FirstName and LastName. You do this by creating a new tuple type for both lists. Using the into clause,

the result from the second collection is put into the variable `yearResults`. `yearResults` is created for every racer in the first collection and contains the results of the matching first name and last name from the second collection. Finally, with the LINQ query, a new tuple type is created that contains the needed information (code file `EnumerableSample/JoinSamples.cs`):

```
static void GroupJoin()
{
  //...
  var q = (from r in Formula1.GetChampions()
           join r2 in racers on
           (
             r.FirstName,
             r.LastName
           )
           equals
           (
             r2.FirstName,
             r2.LastName
           )
           into yearResults
           select
           (
             r.FirstName,
             r.LastName,
             r.Wins,
             r.Starts,
             Results: yearResults
           ));

  foreach (var r in q)
  {
    Console.WriteLine($"{r.FirstName} {r.LastName}");
    foreach (var results in r.Results)
    {
      Console.WriteLine($"\t{results.Year} {results.Position}");
    }
  }
}
```

The last results from the `foreach` loop are shown next. Jenson Button has been among the top three for three years—in 2004 as third, 2009 as first, and 2011 as second; Sebastian Vettel was world champion four times, had the second position three times, and the third in 2015; and Nico Rosberg was world champion in 2016 and in the second position two times:

```
Jenson Button
        2004 3
        2009 1
        2011 2
Sebastian Vettel
        2009 2
        2010 1
        2011 1
        2012 1
        2013 1
        2015 3
        2017 2
        2018 2
```

```
Nico Rosberg
        2014 2
        2015 2
        2016 1
```

Using `GroupJoin` with extension methods, the syntax probably looks a bit easier to catch. First, the compound from is done with the `SelectMany` method. This part is not very different, and tuples are used again. The `GroupJoin` method is invoked by passing the racers with the first parameter to join the champions with the flattened racers, and the match for both collections with the second and third parameters. The fourth parameter receives the racer from the first collection and a collection of the second. These are the results containing the position and the year which are written to the `Results` tuple member (code file `EnumerableSample/Program.cs`):

```
static void GroupJoinWithMethods()
{
  var racers = Formula1.GetChampionships()
    .SelectMany(cs => new List<(int Year, int Position, string FirstName,
       string LastName)>
    {
      (cs.Year, Position: 1, FirstName: cs.First.FirstName(),
        LastName: cs.First.LastName()),
      (cs.Year, Position: 2, FirstName: cs.Second.FirstName(),
        LastName: cs.Second.LastName()),
      (cs.Year, Position: 3, FirstName: cs.Third.FirstName(),
        LastName: cs.Third.LastName())
    });

  var q = Formula1.GetChampions()
    .GroupJoin(racers,
      r1 => (r1.FirstName, r1.LastName),
      r2 => (r2.FirstName, r2.LastName),
      (r1, r2s) => (r1.FirstName, r1.LastName, r1.Wins, r1.Starts,
        Results: r2s));
  //...
}
```

Set Operations

The extension methods `Distinct`, `Union`, `Intersect`, and `Except` are *set* operations. The following example creates a sequence of Formula 1 champions driving a Ferrari and another sequence of Formula 1 champions driving a McLaren and then determines whether any driver has been a champion driving both of these cars. Of course, that's where the `Intersect` extension method can help.

First, you need to get all champions driving a Ferrari. This uses a simple LINQ query with a compound `from` to access the property `Cars` that's returning a sequence of string objects:

```
var ferrariDrivers = from r in Formula1.GetChampions()
                     from c in r.Cars
                     where c == "Ferrari"
                     orderby r.LastName
                     select r;
```

Now the same query with a different parameter of the `where` clause is needed to get all McLaren racers. It's not a good idea to write the same query again. Another option is to create a method in which you can pass the parameter `car`. In case the method wouldn't be needed in other places, you can create a local function. `racersByCar` is the name of a local function that is implemented as a lambda expression containing a LINQ query. The local function `racersByCar` is defined within the scope of the method `SetOperations`, and thus it can only be invoked

within this method. The LINQ `Intersect` extension method is used to get all racers who won the championship with a Ferrari and a McLaren (code file `EnumerableSample/LinqSamples.cs`):

```
static void SetOperations()
{
  IEnumerable<Racer> racersByCar(string car) =>
    from r in Formula1.GetChampions()
    from c in r.Cars
    where c == car
    orderby r.LastName
    select r;

  Console.WriteLine("World champion with Ferrari and McLaren");
  foreach (var racer in racersByCar("Ferrari").Intersect(racersByCar("McLaren")))
  {
    Console.WriteLine(racer);
  }
}
```

The result is just one racer, Niki Lauda:

```
World champion with Ferrari and McLaren
Niki Lauda
```

> **NOTE** *The set operations compare the objects by invoking the* `GetHashCode` *and* `Equals` *methods of the entity class. For custom comparisons, you can also pass an object that implements the interface* `IEqualityComparer<T>`*. In the preceding example, the* `GetChampions` *method always returns the same objects, so the default comparison works. If that's not the case, the set methods offer overloads in which a comparison can be defined.*

Zip

The `Zip` method enables you to merge two related sequences into one with a predicate function.

First, two related sequences are created, both with the same filtering (country Italy) and ordering. For merging, this is important because item 1 from the first collection is merged with item 1 from the second collection, item 2 with item 2, and so on. In case the count of the two sequences is different, `Zip` stops when the end of the smaller collection is reached.

The items in the first collection have a `Name` property, and the items in the second collection have `LastName` and `Starts` properties.

Using the `Zip` method on the collection, `racerNames` requires the second collection `racerNamesAndStarts` as the first parameter. The second parameter is of type `Func<TFirst, TSecond, TResult>`. This parameter is implemented as a lambda expression and receives the elements of the first collection with the parameter `first`, and the elements of the second collection with the parameter `second`. The implementation creates and returns a string containing the `Name` property of the first element and the `Starts` property of the second element (code file `EnumerableSample/LinqSamples.cs`):

```
static void ZipOperation()
{
  var racerNames = from r in Formula1.GetChampions()
                   where r.Country == "Italy"
                   orderby r.Wins descending
```

```
                    select new
                    {
                      Name = r.FirstName + " " + r.LastName
                    };

    var racerNamesAndStarts = from r in Formula1.GetChampions()
                              where r.Country == "Italy"
                              orderby r.Wins descending
                              select new
                              {
                                r.LastName,
                                r.Starts
                              };

    var racers = racerNames.Zip(racerNamesAndStarts,
                  (first, second) => first.Name + ", starts: " + second.Starts);

    foreach (var r in racers)
    {
      Console.WriteLine(r);
    }
  }
```

The result of this merge is shown here:

```
Alberto Ascari, starts: 32
Nino Farina, starts: 33
```

Partitioning

Partitioning operations such as the extension methods `Take` and `Skip` can be used for easy paging—for example, to display just five racers on the first page, and continue with the next five on the following pages.

With the LINQ query shown here, the extension methods `Skip` and `Take` are added to the end of the query. The `Skip` method first ignores a number of items calculated based on the page size and the actual page number; the `Take` method then takes a number of items based on the page size (code file `EnumerableSample/LinqSamples.cs`):

```
public static void Partitioning()
{
  int pageSize = 5;
  int numberPages = (int)Math.Ceiling(Formula1.GetChampions().Count() /
    (double)pageSize);

  for (int page = 0; page < numberPages; page++)
  {
    Console.WriteLine($"Page {page}");

    var racers = (from r in Formula1.GetChampions()
                  orderby r.LastName, r.FirstName
                  select r.FirstName + " " + r.LastName)
                  .Skip(page * pageSize).Take(pageSize);

    foreach (var name in racers)
    {
      Console.WriteLine(name);
```

```
        }
        Console.WriteLine();
      }
    }
```

Here is the output of the first three pages:

```
Page 0
Fernando Alonso
Mario Andretti
Alberto Ascari
Jack Brabham
Jenson Button

Page 1
Jim Clark
Juan Manuel Fangio
Nino Farina
Emerson Fittipaldi
Mika Hakkinen

Page 2
Lewis Hamilton
Mike Hawthorn
Damon Hill
Graham Hill
Phil Hill
```

Paging can be extremely useful with Windows or web applications for showing the user only a part of the data.

> **NOTE** *This paging mechanism has an important behavior: Because the query is done with every page, changing the underlying data affects the results (for example when accessing a database). New objects are shown as paging continues. Depending on your scenario, this can be advantageous to your application. If this behavior is not what you need, you can do the paging not over the original data source but by using a cache that maps to the original data.*

With the `TakeWhile` and `SkipWhile` extension methods, you can also pass a predicate to retrieve or skip items based on the result of the predicate.

Aggregate Operators

The aggregate operators such as `Count`, `Sum`, `Min`, `Max`, `Average`, and `Aggregate` do not return a sequence; instead they return a single value.

The `Count` extension method returns the number of items in the collection. In the following example, the `Count` method is applied to the `Years` property of a `Racer` to filter the racers and return only those who won more than three championships. Because the same count is needed more than once in the same query, a variable `numberYears` is defined by using the `let` clause (code file `EnumerableSample/LinqSamples.cs`):

```
static void AggregateCount()
{
  var query = from r in Formula1.GetChampions()
              let numberYears = r.Years.Count()
              where numberYears >= 3
```

```
                    orderby numberYears descending, r.LastName
                    select new
                    {
                      Name = r.FirstName + " " + r.LastName,
                      TimesChampion = numberYears
                    };

            foreach (var r in query)
            {
              Console.WriteLine($"{r.Name} {r.TimesChampion}");
            }
          }
```

The result is shown here:

```
Michael Schumacher 7
Lewis Hamilton 6
Juan Manuel Fangio 5
Alain Prost 4
Sebastian Vettel 4
Jack Brabham 3
Niki Lauda 3
Nelson Piquet 3
Ayrton Senna 3
Jackie Stewart 3
```

The `Sum` method summarizes all numbers of a sequence and returns the result. In the next example, `Sum` is used to calculate the sum of all race wins for a country. First the racers are grouped based on country; then, with the new anonymous type created, the `Wins` property is assigned to the sum of all wins from a single country (code file `EnumerableSample/Program.cs`):

```
static void AggregateSum()
{
  var countries = (from c in
                   from r in Formula1.GetChampions()
                   group r by r.Country into c
                   select new
                   {
                     Country = c.Key,
                     Wins = (from r1 in c
                             select r1.Wins).Sum()
                   }
                   orderby c.Wins descending, c.Country
                   select c).Take(5);

  foreach (var country in countries)
  {
    Console.WriteLine($"{country.Country} {country.Wins}");
  }
}
```

The most successful countries based on the Formula 1 race champions are as follows:

```
UK 245
Germany 168
Brazil 78
France 51
Finland 46
```

The methods `Min`, `Max`, `Average`, and `Aggregate` are used in the same way as `Count` and `Sum`. `Min` returns the minimum number of the values in the collection, and `Max` returns the maximum number. `Average` calculates the average number. With the `Aggregate` method, you can pass a lambda expression that performs an aggregation of all the values.

Conversion Operators

In this chapter, you've already seen that query execution is deferred until the items are accessed. Using the query within an iteration, the query is executed. With a conversion operator, the query is executed immediately and the result is returned in an array, a list, or a dictionary.

In the next example, the `ToList` extension method is invoked to immediately execute the query and put the result into a `List<T>` (code file `EnumerableSample/LinqSamples.cs`):

```
static void ToList()
{
  List<Racer> racers = (from r in Formula1.GetChampions()
                        where r.Starts > 220
                        orderby r.Starts descending
                        select r).ToList();

  foreach (var racer in racers)
  {
    Console.WriteLine($"{racer} {racer:S}");
  }
}
```

The result of this query shows Jenson Button first:

```
Kimi Räikkönen 323
Fernando Alonso 314
Jenson Button 306
Michael Schumacher 287
Lewis Hamilton 260
Sebastian Vettel 250
```

It's not always that simple to get the returned objects into the list. For example, for fast access from a car to a racer within a collection class, you can use the new class `Lookup<TKey, TElement>`.

> **NOTE** *The* `Dictionary<TKey, TValue>` *class supports only a single value for a key. With the class* `Lookup<TKey, TElement>` *from the namespace* `System.Linq`*, you can have multiple values for a single key. These classes are covered in detail in Chapter 8, "Collections."*

In the following example when you use the compound `from` query, the sequence of racers and cars is flattened, and an anonymous type with the properties `Car` and `Racer` is created. With the lookup that is returned, the key should be of type `string` referencing the car, and the value should be of type `Racer`. To make this selection, you can pass a key and an element selector to one overload of the `ToLookup` method. The key selector references the `Car` property, and the element selector references the `Racer` property:

```
public static void ToLookup()
{
  var racers = (from r in Formula1.GetChampions()
                from c in r.Cars
```

```
                select new
                {
                  Car = c,
                  Racer = r
                }).ToLookup(cr => cr.Car, cr => cr.Racer);

  foreach (var williamsRacer in racers["Williams"])
  {
    Console.WriteLine(williamsRacer);
  }
}
```

The result of all "Williams" champions accessed using the indexer of the `Lookup` class is shown here:

```
Alan Jones
Keke Rosberg
Nelson Piquet
Nigel Mansell
Alain Prost
Damon Hill
Jacques Villeneuve
```

In case you need to use a LINQ query over an untyped collection, such as the `ArrayList`, you can use the `Cast` method. In the following example, an `ArrayList` collection that is based on the `Object` type is filled with `Racer` objects. To make it possible to define a strongly typed query, you can use the `Cast` method:

```
public static void ConvertWithCast()
{
  var list = new System.Collections.ArrayList(Formula1.GetChampions()
    as System.Collections.ICollection);

  var query = from r in list.Cast<Racer>()
              where r.Country == "USA"
              orderby r.Wins descending
              select r;

  foreach (var racer in query)
  {
    Console.WriteLine($"{racer:A}");
  }
}
```

The results include the only Formula 1 champion from the United States:

```
Mario Andretti, country: USA, starts: 128, wins: 12
Phil Hill, country: USA, starts: 48, wins: 3
```

Generation Operators

The generation operators `Range`, `Empty`, and `Repeat` are not extension methods but normal static methods that return sequences. With LINQ to Objects, these methods are available with the `Enumerable` class.

Have you ever needed a range of numbers filled? Nothing is easier than using the `Range` method. This method receives the start value with the first parameter and the number of items with the second parameter:

```
static void GenerateRange()
{
```

```
  var values = Enumerable.Range(1, 20);
  foreach (var item in values)
  {
    Console.Write($"{item} ", item);
  }
  Console.WriteLine();
}
```

> **NOTE** *The* `Range` *method does not return a collection filled with the values as defined. This method does a deferred query execution similar to the other methods. It returns a* `RangeEnumerator` *that simply does a* `yield return` *with the values incremented.*

Of course, the result now looks like this:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

You can combine the result with other extension methods to get a different result—for example, using the `Select` extension method:

```
var values = Enumerable.Range(1, 20).Select(n => n * 3);
```

The `Empty` method returns an iterator that does not return values. This can be used for parameters that require a collection for which you can pass an empty collection.

The `Repeat` method returns an iterator that returns the same value a specific number of times.

PARALLEL LINQ

The class `ParallelEnumerable` in the `System.Linq` namespace splits the work of queries across multiple threads that run simultaneously on multiple processors. Although the `Enumerable` class defines extension methods to the `IEnumerable<T>` interface, most extension methods of the `ParallelEnumerable` class are extensions for the class `ParallelQuery<TSource>`. One important exception is the `AsParallel` method, which extends `IEnumerable<TSource>` and returns `ParallelQuery<TSource>`, so a normal collection class can be queried in a parallel manner.

Parallel Queries

To demonstrate Parallel LINQ (PLINQ), a large collection is needed. With small collections, you don't see any effect when the collection fits inside the CPU's cache. In the following code, a large `int` collection is filled with random values (code file `ParallelLinqSample/Program.cs`):

```
static IEnumerable<int> SampleData()
{
  const int arraySize = 500_00_000;
  var r = new Random();
  return Enumerable.Range(0, arraySize).Select(x => r.Next(140)).ToList();
}
```

Now you can use a LINQ query to filter the data, do some calculations, and get an average of the filtered data. The query defines a filter with the `where` clause to summarize only the items with values whose natural logarithm

is less than four, and then the aggregation function `Average` is invoked. The only difference from the LINQ queries you've seen so far is the call to the `AsParallel` method:

```
static void LinqQuery(IEnumerable<int> data)
{
  var res = (from x in data.AsParallel()
             where Math.Log(x) < 4
             select x).Average();
  //...
}
```

Like the LINQ queries shown already, the compiler changes the syntax to invoke the methods `AsParallel`, `Where`, `Select`, and `Average`. `AsParallel` is defined with the `ParallelEnumerable` class to extend the `IEnumerable<T>` interface, so it can be called with a simple array. `AsParallel` returns `ParallelQuery<TSource>`. Because of the returned type, the `Where` method chosen by the compiler is `ParallelEnumerable.Where` instead of `Enumerable.Where`.

In the following code, the `Select` and `Average` methods are from `ParallelEnumerable` as well. In contrast to the implementation of the `Enumerable` class, with the `ParallelEnumerable` class, the query is *partitioned* so that multiple threads can work on the query. The collection can be split into multiple parts whereby different threads work on each part to filter the remaining items. After the partitioned work is completed, *merging* must occur to get the summary result of all parts:

```
static void ExtensionMethods(IEnumerable<int> data)
{
  var res = data.AsParallel()
    .Where(x => Math.Log(x) < 4)
    .Select(x => x).Average();
  //...
}
```

When you run this code, you can also start the task manager so you can confirm that all CPUs of your system are busy. If you remove the `AsParallel` method, multiple CPUs might not be used. Of course, if you don't have multiple CPUs on your system, then don't expect to see an improvement with the parallel version. On my system with 8 logical processors, the method runs 0.351 seconds with `AsParallel` and 1.23 seconds without.

> **NOTE** *You can customize parallel queries using extension methods such as* `WithExecutionMode`, `WithDegreeOfParallelism`, *and even custom partitioners. With* `WithExecutionMode` *you can pass a value of* `ParallelExecutionMode`, *which can be* `Default` *or* `ForceParallelism`. *By default, Parallel LINQ avoids parallelism with high overhead. With the method* `WithDegreeOfParallelism`, *you can pass an integer value to specify the maximum number of tasks that should run in parallel. This is useful if not all CPU cores should be used by the query. .NET contains specific partitioners that are used based on the collection types, e.g. for arrays, and the generic List type. You might have different requirements, or specific collection types with other layouts that could take advantage of partitioning of the data in other ways. Here you can write a custom partitioner deriving from the generic base classes* `OrderablePartitioner` *or* `Partitioner` *in the namespace* `System.Collections.Generic`.

Cancellation

.NET offers a standard way to cancel long-running tasks, and this is also true for Parallel LINQ. The classes needed for cancellation are defined in the `System.Threading` namespace.

To cancel a long-running query, you can add the method `WithCancellation` to the query and pass a `CancellationToken` to the parameter. The `CancellationToken` is created from the `CancellationTokenSource` instance that you create. The query is run in a separate thread where the exception of type `OperationCanceledException` is caught. This exception is fired if the query is canceled. From the main thread, the task can be canceled by invoking the `Cancel` method of the `CancellationTokenSource` (code file `ParallelLinqSample/Program.cs`):

```
public static void UseCancellation(IEnumerable<int> data)
{
  CancellationTokenSource cts = new();

  Task.Run(() =>
  {
    try
    {
      var res = (from x in data.AsParallel().WithCancellation(cts.Token)
                 where Math.Log(x) < 4
                 select x).Average();

      Console.WriteLine($"Query finished, sum: {res}");
    }
    catch (OperationCanceledException ex)
    {
      Console.WriteLine(ex.Message);
    }
  });
  Console.WriteLine("Query started");
  Console.Write("Cancel? ");
  string input = Console.ReadLine();
  if (input.ToLower().Equals("y"))
  {
    cts.Cancel();
  }
}
```

> **NOTE** *You can read more about cancellation and the* `CancellationToken` *in Chapter 17, "Parallel Programming."*

EXPRESSION TREES

With LINQ to Objects, the extension methods require a delegate type as parameter; this way, a lambda expression can be assigned to the parameter. Lambda expressions can also be assigned to parameters of type `Expression<T>`. The C# compiler defines different behavior for lambda expressions depending on the type. If the type is `Expression<T>`, the compiler creates an expression tree from the lambda expression and stores it in the assembly. The expression tree can be analyzed during runtime and optimized for querying against the data source.

Consider the following query:

```
var brazilRacers = from r in Formula1.GetChampions()
                   where r.Country == "Brazil"
                   orderby r.Wins descending
                   select r;
```

The preceding query expression uses the extension methods `Where`, `OrderByDescending`, and `Select`. The `Enumerable` class defines the `Where` extension method with the delegate type `Func<T, bool>` as a parameter predicate:

```
public static IEnumerable<TSource> Where<TSource>(
  this IEnumerable<TSource> source, Func<TSource, bool> predicate);
```

This way, the lambda expression is assigned to the predicate. Here, the lambda expression is similar to an anonymous method, as explained earlier:

```
Func<Racer, bool> predicate = r => r.Country == "Brazil";
```

The `Enumerable` class is not the only class for defining the `Where` extension method. The `Where` extension method is also defined by the class `Queryable<T>`. This class has a different definition of the `Where` extension method:

```
public static IQueryable<TSource> Where<TSource>(
  this IQueryable<TSource> source,
  Expression<Func<TSource, bool>> predicate);
```

Here, the lambda expression is assigned to the type `Expression<T>` (namespace `System.Linq.Expressions`) which behaves differently:

```
Expression<Func<Racer, bool>> predicate = r => r.Country == "Brazil";
```

Instead of using delegates, the compiler emits an expression tree to the assembly. The expression tree can be read during runtime. Expression trees are built from classes derived from the abstract base class `Expression`. The `Expression` class is not the same as `Expression<T>`. Some of the expression classes that inherit from `Expression` include `BinaryExpression`, `ConstantExpression`, `InvocationExpression`, `LambdaExpression`, `NewExpression`, `NewArrayExpression`, `TernaryExpression`, `UnaryExpression`, and more. The compiler creates an expression tree resulting from the lambda expression.

For example, the lambda expression `r.Country == "Brazil"` makes use of `ParameterExpression`, `MemberExpression`, `ConstantExpression`, and `MethodCallExpression` to create a tree and store the tree in the assembly. This tree is then used during runtime to create an optimized query to the underlying data source.

With the sample application, the method `DisplayTree` is implemented to display an expression tree graphically on the console. In the following example, an `Expression` object can be passed, and depending on the expression type, some information about the expression is written to the console. Depending on the type of the expression, `DisplayTree` is called recursively (code file `ExpressionTreeSample/Program.cs`):

```
static void DisplayTree(int indent, string message,
  Expression expression)
{
  string output = $"{string.Empty.PadLeft(indent, '>')} {message}" +
    $"! NodeType: {expression.NodeType}; Expr: {expression}";

  indent++;

  switch (expression.NodeType)
  {
    case ExpressionType.Lambda:
      Console.WriteLine(output);
      LambdaExpression lambdaExpr = (LambdaExpression)expression;
      foreach (var parameter in lambdaExpr.Parameters)
      {
        DisplayTree(indent, "Parameter", parameter);
      }
```

```
        DisplayTree(indent, "Body", lambdaExpr.Body);
        break;
      case ExpressionType.Constant:
        ConstantExpression constExpr = (ConstantExpression)expression;
        Console.WriteLine($"{output} Const Value: {constExpr.Value}");
        break;
      case ExpressionType.Parameter:
        ParameterExpression paramExpr = (ParameterExpression)expression;
        Console.WriteLine($"{output} Param Type: {paramExpr.Type.Name}");
        break;
      case ExpressionType.Equal:
      case ExpressionType.AndAlso:
      case ExpressionType.GreaterThan:
        BinaryExpression binExpr = (BinaryExpression)expression;
        if (binExpr.Method != null)
        {
          Console.WriteLine($"{output} Method: {binExpr.Method.Name}");
        }
        else
        {
          Console.WriteLine(output);
        }
        DisplayTree(indent, "Left", binExpr.Left);
        DisplayTree(indent, "Right", binExpr.Right);
        break;
      case ExpressionType.MemberAccess:
        MemberExpression memberExpr = (MemberExpression)expression;
        Console.WriteLine($"{output} Member Name: {memberExpr.Member.Name}, " +
          " Type: {memberExpr.Expression}");
        DisplayTree(indent, "Member Expr", memberExpr.Expression);
        break;
      default:
        Console.WriteLine();
        Console.WriteLine($"{expression.NodeType} {expression.Type.Name}");
        break;
    }
  }
```

> **NOTE** *The method* `DisplayTree` *does not deal with all expression types—only the types that are used with the following example expression.*

The expression that is used for showing the tree is already well known. It's a lambda expression with a `Racer` parameter, and the body of the expression takes racers from Brazil only if they have won more than six races. This expression is passed to the `DisplayTree` method to see the tree:

```
Expression<Func<Racer, bool>> expression =
  r => r.Country == "Brazil" && r.Wins > 6;

DisplayTree(0, "Lambda", expression);
```

Looking at the tree result, you can see from the output that the lambda expression consists of a `Parameter` and an `AndAlso` node type. The `AndAlso` node type has an `Equal` node type to the left and a `GreaterThan` node type to the right. The `Equal` node type to the left of the `AndAlso` node type has a `MemberAccess` node type to the left and a `Constant` node type to the right, and so on:

```
Lambda! NodeType: Lambda; Expr: r => ((r.Country == "Brazil") AndAlso (r.Wins > 6))
> Parameter! NodeType: Parameter; Expr: r Param Type: Racer
> Body! NodeType: AndAlso; Expr: ((r.Country == "Brazil") AndAlso (r.Wins > 6))
>> Left! NodeType: Equal; Expr: (r.Country == "Brazil") Method: op_Equality
>>> Left! NodeType: MemberAccess; Expr: r.Country Member Name: Country, Type: String
>>>> Member Expr! NodeType: Parameter; Expr: r Param Type: Racer
>>> Right! NodeType: Constant; Expr: "Brazil" Const Value: Brazil
>> Right! NodeType: GreaterThan; Expr: (r.Wins > 6)
>>> Left! NodeType: MemberAccess; Expr: r.Wins Member Name: Wins, Type: Int32
>>>> Member Expr! NodeType: Parameter; Expr: r Param Type: Racer
>>> Right! NodeType: Constant; Expr: 6 Const Value: 6
```

Entity Framework Core (EF Core) is an example where the `Expression<T>` type is used. EF Core providers convert LINQ expression trees to SQL statements.

LINQ PROVIDERS

.NET includes several LINQ providers. A LINQ provider implements the standard query operators for a specific data source. LINQ providers might implement more extension methods than are defined by LINQ, but the standard operators must at least be implemented. LINQ to XML implements additional methods that are particularly useful with XML, such as the methods `Elements`, `Descendants`, and `Ancestors` defined by the class `Extensions` in the `System.Xml.Linq` namespace.

Implementation of the LINQ provider is selected based on the namespace and the type of the first parameter. The namespace of the class that implements the extension methods must be opened; otherwise, the extension class is not in scope. The parameter of the `Where` method defined by LINQ to Objects and the `Where` method defined by LINQ to Entities is different.

The `Where` method of LINQ to Objects is defined with the `Enumerable` class:

```
public static IEnumerable<TSource> Where<TSource>(
  this IEnumerable<TSource> source, Func<TSource, bool> predicate);
```

Inside the `System.Linq` namespace is another class that implements the operator `Where`. This implementation is used by LINQ to Entities. You can find the implementation in the class `Queryable`:

```
public static IQueryable<TSource> Where<TSource>(
  this IQueryable<TSource> source,
  Expression<Func<TSource, bool>> predicate);
```

Both of these classes are implemented in the `System.Core` assembly in the `System.Linq` namespace. How does the compiler select what method to use, and what's the magic in the `Expression` type? The lambda expression is the same regardless of whether it is passed with a `Func<TSource, bool>` parameter or an `Expression<Func<TSource, bool>>` parameter—only the compiler behaves differently. The selection is done based on the `source` parameter. The method that matches best based on its parameters is chosen by the compiler. Properties of Entity Framework Core contexts are of type `DbSet<TEntity>`. `DbSet<TEntity>` implements `IQueryable<TEntity>`, and thus Entity Framework Core uses the `Where` method of the `Queryable` class.

SUMMARY

This chapter described and demonstrated the LINQ query and the language constructs on which the query is based, such as extension methods and lambda expressions. You've looked at the various LINQ query operators—not only for filtering and ordering of data sources, but also for partitioning, grouping, doing conversions, joins, and so on.

With Parallel LINQ, you've seen how longer queries can easily be parallelized.

Another important concept of this chapter is the expression tree. Expression trees allow a program to build the tree at compile time, store it in the assembly, and then optimize it at runtime. You can read about its great advantages in Chapter 21.

The next chapter covers error handling—getting into the `try`, `catch`, and `throw` keywords.